Practical Correctness and Equivalence Checking for MLIR

Emily Tucker
Georgia Institute of Technology
emily.tucker@gatech.edu

Louis-Noël Pouchet
Colorado State University
pouchet@colostate.edu

Erika Hunhoff
AMD
erika.hunhoff@amd.com

Stephen Neuendorffer
AMD
stephen.neuendorffer@amd.com

Erwei Wang
AMD
erwei.wang@amd.com

Abstract

The typical development flow for compilation of programs onto parallel architectures proceeds by successive analysis and transformations of the program, progressively lowering the program semantics to machine-level instructions. LLVM's Multi-Level Intermediate Representation (MLIR) ecosystem provides natural building blocks to implement such a compilation flow: for example, starting from a simple sequential description of a linear algebra operation in the `linalg` dialect, the program is progressively optimized via tiling and parallelization using specific dialects such as `affine` to facilitate the development of such automatic transformations. Then, target-specific compilation flows can be developed. The AMD MLIR-AIR and MLIR-AIE compilers use this progressive lowering approach to map computations onto the AI engine neural accelerator. In practice, for important workloads, optimizations may also be manually implemented and tuned for additional performance.

In this work, we target the correctness verification of these modern compilation flows, integrating seamlessly at any phase of this MLIR optimizing and lowering process. We present PEQC-MLIR, which can prove equivalence between two MLIR programs. PEQC-MLIR enables users to quickly check for correctness and possible regressions of a (parallelizing) compiler, assists developers in implementing compilers as well as manual optimizations, and alleviates drawbacks of testing by computing equivalence using symbolic inputs. We evaluate PEQC-MLIR on thousands of MLIR variants of PolyBench/C generated using MLIR tools, uncovering incorrectness in raising and optimization tools. We evaluate PEQC-MLIR on the MLIR-AIR and MLIR-AIE toolchain, exposing subtle concurrency bugs in the MLIR-AIE toolchain that eluded testing.

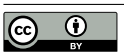
CCS Concepts

• **Software and its engineering → Formal software verification**.

Keywords

MLIR, program equivalence, formal verification

ACM Reference Format:
Emily Tucker, Louis-Noël Pouchet, Erika Hunhoff, Stephen Neuendorffer, and Erwei Wang. 2026. Practical Correctness and Equivalence Checking for MLIR. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19–22, 2026, Chicago, IL, USA*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3838684.3846878



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846878

1 Introduction

The design of compiler optimizations is typically organized into passes, successively applied in a pipeline to progressively optimize and lower the program to machine-level representation. LLVM exemplifies this model, and facilitates the design and reuse of optimization passes by providing a unified, common representation for input and output of these passes: the SSA-based LLVM IR. More recently, the LLVM MLIR approach [5, 44] transformed this paradigm. Typically, in each pass that requires information such as coarse-grain parallelization [5] or polyhedral optimization [18], reconstructing high-level information, such as loop structure, is needed. Instead, MLIR proposes lowering between a variety of composable program representations called *dialects*. This enables the design of MLIR dialects (languages defining an IR) targeting, e.g., linear algebra computations (`linalg`), affine programs (`affine`) or simply structured control-flow (`scf`). This critical flexibility of the representation abstraction level enables the design of myriad dialect-centric optimizations: for example, tiling and parallelization of affine programs, or target-optimized transformations for key linear algebra workloads prevalent in current AI deployment.

For example, when targeting acceleration of AI workloads, a typical workflow starts from a PyTorch or equivalent high-level representation of the algorithm, which is then transformed into, e.g., `linalg`. The AMD MLIR-AIR [4] dialect, for instance, takes `linalg` as input and targets spatial accelerators implemented as communicating Processing Elements (PEs), automatically extracting and exploiting parallelism in the program. The MLIR-AIE dialect [3] is then used to lower AIR programs into ones specific to the AMD AI Engine hardware design. An LLVM backend is used to generate binary code for individual processors, while binary code for other components, such as DMA Engines, is generated directly from the MLIR representation.

Overall, the number of compilers and intermediate representations (dialects) deployed in production keeps increasing, enabled by MLIR. Assessing the correctness of these compilers remains an open and increasingly *varied* challenge. In this work, we target the correctness verification of transformed MLIR programs in these modern flows. Taking as input a pair of programs, one being a transformed version of the other, we aim to *prove* their semantic equivalence. We target a practical class of programs that includes highly optimized concurrent implementations of key AI

workloads for spatial accelerators, supporting a range of MLIR dialects from `linalg` down to lower-level hardware-centric dialects like MLIR-AIE. We achieve this coverage by introducing a system that *automatically and faithfully lowers MLIR programs to a subset of standard dialects*, and then *determines semantic equivalence* between two such lowered programs.

Testing is typically deployed to check correctness (e.g., by checking that the outputs produced by two programs are identical), scaling to complex workloads. However, testing alone does not provide any guarantee of correctness, merely the absence of some visible bugs. To prove correctness for any input, approaches to formal correctness verification and equivalence, e.g., [30, 33, 38, 41, 42, 46, 48, 51, 57, 59, 63, 76, 79] may be deployed by reasoning on symbolic inputs. But, in turn, their time-to-solution can be prohibitive, even for the simple transformations of concurrent control-flow we target. In this work, we propose a novel solution: our formal verification approach operates in time linear in the number of operations executed by the program and proves correctness on symbolic inputs. We define and handle a rich class of deterministic concurrent programs with our approach. It scales to realistic end-to-end designs for complex spatial accelerators, and can assess correctness of a myriad of `mlir-opt` production optimizations in minutes.

Specifically, we leverage the approach developed in PEQC [56] to propose a novel framework for verification via hybrid concrete-symbolic interpretation of concurrent MLIR programs. We significantly revisit and extend PEQC in PEQC-MLIR, the system presented in this paper. PEQC-MLIR supports verification of: (1) general, hierarchical asynchronous tasks, (2) arbitrary synchronizations with binary (`wait`/`set`) and counting (`acquire`/`release`) semaphores, and (3) a class of symbolic conditional control-flow, significantly broadening the scope of applicability of this approach. The correctness of PEQC-MLIR's approach is formally proved in [69], and its whole implementation, including the lowering passes we develop and rely on, is part of the trusted codebase. We make the following contributions:

- We introduce PEQC-MLIR, our approach to the correctness verification of a rich class of MLIR programs and their transformations, by relying on successive lowering and conversion of user-defined dialects to a subset of standard dialects. PEQC-MLIR provides formal correctness verification and equivalence checking for a practical class of sequential and concurrent MLIR programs.
- We introduce PolyBenchEq/MLIR, a verified correct MLIR port of PolyBench/C, along with numerous equivalent and non-equivalent variants enabling the community to evaluate and develop novel MLIR verification approaches.
- We present extensive evaluation of PEQC-MLIR in Sec. 5, over thousands of MLIR program variants generated using production and research tools, demonstrating how PEQC-MLIR can expose bugs, misuse, and lack of coverage in optimization tools in minutes. We present our verification for the AMD MLIR-AIR and MLIR-AIE toolchains for mapping AI workloads to the AI Engine, exposing subtle concurrency bugs which eluded testing.

The paper is organized as follows. Sec. 2 outlines our approach and challenges it targets. Sec. 3 describes our formal verification system to establish equivalence between two programs, and Sec. 4 presents our MLIR lowering and conversion system. Experimental results are presented in Sec. 5 before discussing related work in Sec. 6.

2 Motivation and Overview

We now present a brief overview of the landscape of techniques to check correctness of program transformations, ranging from testing for the absence of obvious bugs to formal proofs of correctness. Related work is discussed at length in Sec. 6. PEQC-MLIR, introduced in this work, aims to *prove* correctness and achieve fast and predictable time-to-solution and *specializes* verification for a class of deterministic concurrent programs with fixed problem sizes.

Testing to expose bugs The most typical approach to assess the presence of a bug is *testing*: running the two programs on a (series of) input data on the target hardware, and comparing whether they produce identical results. Fuzzing can be deployed to vary inputs and increase confidence in exercising control-flow paths in the program [21, 49]. Instrumentation, e.g., to detect races [13, 34, 66] by computing metadata about the program when it is running can inform of concurrency problems for the tested data. But *testing does not provide any guarantee of correctness*. In fact, it may even give a false sense of confidence if tests pass: a program with more concurrency than the hardware tested on will be partially serialized, possibly hiding concurrency bugs. Race detectors are also typically unable to reason on correctness of computations: they only focus on the correctness of *some* parallelization aspects of the program.

Formal correctness checking Ideally, *formal verification* is deployed to prove properties of programs. Translation validation [48, 54], including for MLIR [14, 30, 76], typically targets correctness checking of alternate implementations for lower-level operations (e.g., checking the absence of undefined behaviors or overflow) by reasoning on all possible execution paths and propagation of properties along them using SMT solvers. Other approaches include model checking [33, 38, 70] to reason about a state space for the program, relying on complex SMT queries to capture the control-flow. Symbolic interpretation and execution can reason about program correctness independently of input data and prove equivalence between programs (e.g., UC-KLEE [57]).

Although these techniques offer the expected *formal guarantee* of correctness and target general-purpose programs in theory, they have typically been very limited in the scope and complexity of transformations they can manipulate. For example, UC-KLEE fails to handle simple loop fusion on gemm using parametric loops, and takes one minute for a simple 2×2 gemm using floats (and times out after 1h for any larger problem) [47]. On the other hand, these approaches *succeed brilliantly* within a narrow scope of properties (e.g., correctness of concurrent data structures [43], bounded translation validation for LLVM [48] and MLIR [30, 76]). However, in our experience, none of these techniques computes a proof of equivalence for the class of floating-point intensive programs and their complex asynchronous concurrent implementation we target: `async` MLIR programs capturing optimized mapping of AI workloads on spatial accelerators.

Overview of PEQC-MLIR In this work, we present PEQC-MLIR, our approach to address the challenges above. PEQC-MLIR is agnostic to how programs are written, and does not need any information about the transformations implemented. It *proves the equivalence of a program and its transformed version, on symbolic inputs*, addressing the limitations of testing. But it does so by very carefully *restricting*

the class of programs it supports, trading off full generality for predictable correctness verification speed.

PEQC-MLIR addresses a fundamental limitation of typical formal verification techniques, by *scaling linearly with the number of operations executed in the program*, at a typical throughput of about 0.5 million statements per second. Although this remains 1000x slower than testing, it *proves* correctness. PEQC-MLIR supports the verification of concurrent programs which can be expressed using hierarchical asynchronous tasks, synchronized with binary and counting semaphores. This verification model is general enough to represent numerous parallel language constructs from MLIR dialects which can be lowered to this model as shown in Sec. 4.

This linear complexity is simply not achievable by typical verification techniques, especially in light of the highly complex concurrency patterns we target, such as those that manifest in compilation for AI accelerators. We achieve this complexity by *targeting a class of deterministic concurrent programs with statically interpretable control-flow*, a generalization of PEQC [56]. In essence, approaches targeting "general" programs typically suffer a path explosion issue: they *search* (by heuristic, e.g. tactics in Rocq, or by SMT queries e.g. in KLEE) for feasible successors to their proof/interpretation step.

The fundamental insight of PEQC-MLIR is to expose and handle a very rich class of programs where such search is not needed: PEQC-MLIR implements an ultra-specialized search-free technique for symbolic interpretation, enabling its scaling. But in turn, *not all programs are handled, by far.* A deterministic SICF program typically requires problem sizes to be known. For example, PEQC verifies a HLS design for a 32×32-PE systolic array, produced by the AutoSA compiler [75] over 10k lines of code, is equivalent to the 3-line transpose matrix-multiply sequential code in about 29s [56]. Here the PEs process *fixed-size* tiles of a matrix. *But the fixed-size matrices are represented symbolically*: the correctness proof holds for any input of that fixed size, and not only for a specific matrix (as is typical of testing). PEQC-MLIR requires the program to contain enough information to interpret all operations related to control-flow and array subscript computations to a concrete value, to avoid any path exploration. This means a fixed problem size for, e.g., linear algebra computations.

In turn, as demonstrated in Sec. 5, PEQC-MLIR gracefully handles highly concurrent programs in machine-agnostic MLIR-AIR and machine-centric MLIR-AIE dialects for the AMD AI Engine, any affine program with a fixed problem size (PolyBenchEQ/MLIR), and any non-affine program meeting SICF restrictions (e.g., non-affine transformations of affine programs).

3 Approach to Correctness Verification

We now present our approach to prove equivalence between a program and its transformed variant in our framework. We keep the approach of PEQC for building symbolic representations of program computations for equivalence checking [56], but completely revamp and generalize its approach to concurrency correctness verification, enabling arbitrary hierarchical tasks and their synchronizations. Our flow for checking equivalence between two programs is shown in Fig. 1. Sec. 3.1 onward outlines PIR and PEQC's approach to symbolic equivalence. Sec. 3.3 onward presents our approach to correctness checking of concurrent programs supported by PEQC-MLIR, and Sec. 3.5 presents equivalence checking between the representations obtained for two programs.

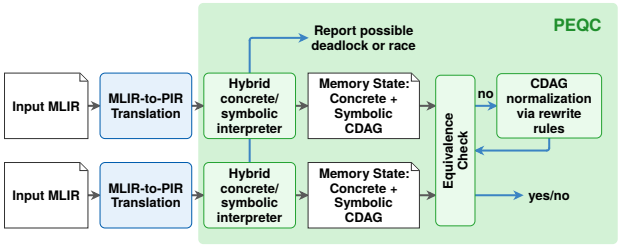


Figure 1: Correctness checking with PEQC-MLIR

3.1 Program Intermediate Representation

PEQC-MLIR builds on PEQC, which takes a program in PIR as input. PIR represents a C program with C semantics for all standard C constructs, but with an additional API made of C functions, implemented by PEQC itself internally, to describe concurrent constructs in the program: spawn async tasks, and semaphore `wait` (blocking) / `set` (non-blocking) and `acquire` (blocking) / `release` (non-blocking). Note, as mentioned above, not all programs in PIR will necessarily supported by PEQC: they may not meet the statically interpretable control-flow restriction [69]. This property can only be ensured by actually (partially) interpreting the program.

3.2 Primer on PEQC

In a nutshell, PEQC [56] builds a symbolic (normalized) representation M of each memory location referenced in the program, and then checks that these representations are equivalent (typically, identical). This equivalence only needs to consider locations accessible after the program terminates, excluding for example local program variables, and accepting a wide variety of program transformations, such as loop transformations and code reassociations. This representation is built by interpreting (parts of) the program. Every time an operation has operands which are computable (e.g., `k = k+1` with `k` having the concrete value 41 at this stage in the program), it is evaluated to a *concrete* value: `k = 42`.

Conversely, if an operation is not computable concretely from the current program state (e.g., because it is a function of some live-in symbolic inputs whose values are not known), then the whole operation is promoted to a symbolic tree-style representation of the computation (e.g., `C[0][0] += A[0][42] * B[42][0]`). Memory locations accessed by the program (e.g., `C[0][0]`) containing symbolic trees are maintained and used as operands as needed to build the next tree using this memory location. Each symbolic tree captures the *complete set of operations and operands used to produce the final value for a memory location.* Technically, this process builds Computation Directed Acyclic Graphs (CDAGs) for the program, which have linear space complexity with respect to the operations executed in the program [26, 39, 56], by carefully deploying shallow copies for symbolic (sub)trees used in multiple destinations.

CDAGs provide a mechanism to understand the behavior of a program for all possible inputs. If the CDAGs for two different programs are isomorphic, then we know that the programs will produce the same output(s) for any possible input(s) [56]. Rewrite rules to normalize and canonicalize the CDAGs, e.g., to implement factorization/distribution may be implemented before comparison, extending the set of programs considered equivalent [56, 69].

However, PEQC can return an interpretation error: it *requires* interpretation to reach the end of the program's control-flow using only concrete values computed during interpretation, and similarly for array subscripts. Consequently, if any control-flow or indexing operation is symbolic, the interpreter fails: the program does not meet the SICF restrictions.

3.3 Programming Model

We now present our programming model for verification of parallel programs. Specifically, we express concurrency as *asynchronous tasks* which are spawned at the time of interpreting the spawn, but may be executed at any time that respects the synchronizations implemented. We support synchronization using semaphores: typically, a semaphore signal is inserted at the end of a task body to alert other tasks waiting on its completion.

PEQC achieves scaling by limiting coverage to programs with a single execution path. PEQC-MLIR achieves scaling for concurrent programs by *limiting coverage to a class of deterministic concurrent programs*: those where interpreting a *single* possible concurrent schedule is sufficient to conclude on *all* such schedules; that is, deterministic concurrent programs [40]. This preserves scaling for correctness verification of concurrent programs we support. We aim to achieve a *sound* conclusion of equivalence (no false positives), irrespective of the latency of operations involved. We therefore develop a model for concurrency checking assuming zero latency for every non-synchronization operation, valid for any possible finite latency for the operations, as shown below.

Definition 3.1 (Zero-latency model). *In a zero-latency model, all operations execute in 0 cycles, and only synchronizations impose a partial order between sets of operations. Given a timestamp for each executed operation $\theta(o) \in \mathbb{N}$, $\theta(o_1) < \theta(o_2)$ iff synchronizations in the program require o_2 to execute after o_1, otherwise $\theta(o_1) = \theta(o_2)$.*

We now define a graph G to encode a zero-latency model for the concurrent program. Our objective is to build G by interpreting the program control-flow, expressing minimal ordering constraints arising from synchronization operations and task spawns.

Definition 3.2 (Happens-before graph G). *G represents the happens-before relation on all operations o in P, in a zero-latency model. There exists a path in G between o_1 and o_2 iff $\theta(o_1) < \theta(o_2)$.*

As shown in Sec. 3.4, we build G incrementally, progressively adding ordering constraints (edges in G) when concurrent operations are interpreted, while maintaining a zero-latency model representation for operations. We define races and deadlocks as follows.

Definition 3.3 (Race). *A race exists in G iff there exist two operations o_1, o_2, executed by two different tasks, that access the same memory location such that $\theta(o_1) = \theta(o_2)$ and either o_1 or o_2 is a write.*

Definition 3.4 (Deadlock). *A deadlock exists iff at least one task is waiting on a semaphore, and no other tasks can make further progress in their execution (i.e. either the tasks have completed, or are also blocked on a semaphore).*

In particular, to build G for asynchronous tasks, PEQC-MLIR builds parts of G for a task sequentially until reaching a blocking semaphore operation, if any. Then, it switches to another active task, and implements a round-robin scheduling between tasks. It revisits a non-completed task every time the semaphore waited on by that task has its value modified by another task.

Consequently, if there is no race or deadlock in a zero-latency model of the program, there cannot be a race or deadlock when considering arbitrary finite latency for any operation.

Theorem 3.5 (Latency-insensitive race and deadlock analysis). *If G does not expose a race or deadlock as per Def. 3.3 and Def. 3.4 in a zero-latency model, then whatever the finite latency of every operation, the program cannot have races or deadlocks.*

Proof. Adding finite latency to operations does not impact the control-flow/dataflow within a task. Intuitively, we can model in G' a latency of x for an operation by replicating its node x times and adding a set/wait synchronization between each. If $\theta(o_1) < \theta(o_2)$ in G, then $\theta(o_1) < \theta(o_2)$ in G'. Conversely, $\theta(o_1) = \theta(o_2)$ in G may not be preserved in G'. Latency can only add edges in G', not remove any: G is a subset of G'. □

PEQC-MLIR can reason about equivalence of a program if it completes interpretation without error and its program graph G contains no races or deadlock, as described in Sec. 3.4.

3.4 Verification of Concurrent Programs

We proceed by conducting hybrid concrete-symbolic interpretation of the program, requiring all operations involved in control-flow and array subscript calculation to be evaluated to concrete values. This process simultaneously builds M, the symbolic representation of the memory state, and G, the program graph, then checks for races and deadlock using G. A program on which interpretation terminates, and at every step preserves determinism in G as described below, is an *α-deterministic program.*

3.4.1 Restrictions on synchronization To build G, we must carefully match operations on the same semaphore: an α-deterministic program may have a series of `set`/`release` operations from various tasks, and a series of `wait`/`acquire` operations from other tasks. From the set of operations on the same semaphore, we must insert the proper edges in G between a `set`/`release` and the corresponding `wait`/`acquire` that consumes it, consistent with the actual partial ordering the synchronization pattern implements.

To make this matching feasible and deterministic, we implement the following restrictions on semaphore use. For every operation interpreted, we augment G accordingly, and check that these properties are satisfied. Otherwise, the program cannot be deterministically interpreted, and PEQC-MLIR outputs an error. Similarly, for every memory access, PEQC-MLIR checks for races using G and fails with error if one is detected. PEQC-MLIR also fails if it detects a deadlock, that is, if no more progress can be made but there are non-completed tasks.

Uniqueness of `set`/`wait` *pairs* We require that one `set` matches exactly one `wait`. That is, no two `set` operations s_1, s_2 on the same semaphore executing at the same $\theta(s_1) = \theta(s_2)$ can be a predecessor in G of a `wait` node. We require that $\theta(s) \leq \theta(w)$ with s : `set`(sem, v) and w : `wait`(sem, v), and that there does not exist s' such that $\theta(s) \leq \theta(s') \leq \theta(w)$ with s' : `set`(sem, v'). We also require the absence of multiple `wait` operations w_1, w_2 on the same semaphore at the same time: $\theta(w_1) \neq \theta(w_2)$.

Restrictions on `release`/`acquire` *pairs* We require that no two `acquire` operations on the same semaphore can happen at the same time. That is, for a_1 and a_2, two `acquires` on the same semaphore, we require $\theta(a_1) \neq \theta(a_2)$. We also require that the quantity released matches the quantity acquired: for every acquire a, the set R of all releases r such that $\theta(r) \leq \theta(a)$ (where r is not matched with another a', where $\theta(r) \leq \theta(a') < \theta(a)$) is the set of a's corresponding releases. The sum of values released in R must not be greater than the value acquired by a.

3.4.2 Determinism The main property that enables our interpretation strategy to work is that interpretation is deterministic. The restrictions in Sec. 3.4.1 ensure both a program's and the interpreter's determinism, so rather than needing to interpret every statement interleaving, the interpreter need only explore one.

To prove determinism of interpretation, we define small-step semantics for program execution that can represent every possible interleaving of tasks. In this model, the only nondeterministic choice is, for tasks that can execute in parallel, which task's next statement is picked to execute next. We prove confluence of this semantics, which means that any choice of next task will result in the same final state, including races and deadlock which result in the error state. The proof relies on every critical pair of parallel steps commuting; the complete proof is presented in [69].

3.4.3 Algorithm Alg. 1 presents a procedure for hybrid concrete-symbolic interpretation of a program. If interpretation succeeds, the program is therefore α-deterministic:

Definition 3.6 (α-deterministic program). *A program that terminates interpretation without error in Alg. 1 is an α-deterministic program.*

This algorithm uses PEQC's multiprogramming approach: all currently active tasks are interpreted until either completing or being blocked by a `wait`/`acquire`. The outer loop on line 6 will return to blocked tasks once another task issues the corresponding `set`/`release`. If no task enables a blocked task to continue, it will be caught as a deadlock on line 43.

During interpretation of a statement, the restrictions on semaphore operations and races are checked. Lines 10–18 check for blocking `waits` and `acquires`, interrupting the task if necessary. Lines 27–40 handle adding edges to G for semaphore synchronization. This algorithm enforces all synchronization restrictions in Sec. 3.4.1, so interpretation is deterministic as per Sec. 3.4.2.

We abstract PEQC's approach to building a concrete/symbolic representation of a statement in the function hybridConcreteSymbolicInterpret(M, s) on line 21 which updates the representation M of all memory locations accessed with the new concrete or symbolic values from the statement s.

Algorithm 1: InterpretConcurrentProgram

```
1  Input: program P
2  Output: memory state M
3  G := ∅ // happens–before graph
4  M := ∅ // memory state
5  T := {⟨getFirstStatement(P), active⟩}; pc := 0
6  while exists active task t in T
7   for active t in T
8    while t active and nextStatementExists(t)
9     s := getNextStatement(t)
10    if s = wait(sem, v) // wait fails
11     if countCorrespondingSemSets(G, pc, sem, v) > 1: ERROR
12     if countCorrespondingSemSets(G, pc, sem, v) = 0:
13      markPaused(t); continue
14    if s = acquire(sem, v) // acquire fails
15     if exists acquire s' where not edgeExists(G, s', s): ERROR
16     sum := sumCorrespondingSemReleases(G, pc, sem)
17     if sum > v: ERROR
18     if sum < v: markPaused(t); continue
19    prevpc := getPrevPC(t); G := G ∪ {prevpc → pc}
20    M, s := hybridConcreteSymbolicInterpret(M, s) // PEQC
21    forall accesses a in s
22     forall a' in prevAccesses(M, loc(a))
23      if !edgeExists(G, pc(a'), pc) and (a or a' is write): ERROR
24    if s = async {P'}
25     t' := ⟨getFirstStatement(P'), active⟩
26     setPrevPC(t', pc); T := T ∪ {t'}
27    else if s = set(sem, v)
28     if exists set s' where edgeExists(G, s', s): ERROR
29     setWaitingTasksToActive(pc, sem, v)
30    else if s = wait(sem, v) // wait succeeds
31     set := getCorrespondingSemSet(G, pc, sem, v)
32     G := G ∪ {set → pc}
33    else if s = release(sem, v)
34     if countCorrespondingSemAcquires(G, pc, sem, v) > 1: ERROR
35     setWaitingAcquiresToActive(sem)
36    else if s = acquire(sem, v) // acquire succeeds
37     sum := sumCorrespondingSemReleases(G, pc, sem)
38     if sum == v
39      for r in getCorrespondingSemReleases(G, pc, sem)
40       G := G ∪ {r → pc}
41    setPrevPC(t, pc); pc ++
42   if !nextStatementExists(t): markComplete(t)
43 if exists t in T not complete: ERROR // deadlock
44 return M
```

3.5 Program Equivalence

If two programs P_A and P_B have reached their final state without error, and the memory computed for each is identical, $M_A = M_B$, then they are equivalent, denoted $P_A \equiv P_B$. We can broaden beyond strict equality by normalizing M before checking for equality, for example by using rewrite rules to normalize symbolic expressions (e.g., $sym(a * b) \equiv sym(b * a)$) [56].

Technically, checking equivalence of local variables (e.g., loop counters) is over-constrained: only non-local variables need to be

computed equivalently, as local variables are not alive when the program terminates. This leads to Theorem 3.7, where we check the union of non-local variables in both programs, to detect side effects appearing in one program but not the other.

Theorem 3.7 (Equivalence of non-local variables). *Let P_A, P_B be two programs on which Alg. 1 terminates without error, with final memory representations M_A, M_B. Let $S = \{v \mid M_A(v) = M_B(v)\}$ be the set of variables which are equivalent. Suppose that V is the union of the non-local variables in P_A and P_B: the non-local memory locations. Then $V \subseteq S$ implies that $P_A \equiv P_B$.*

Proof. Follows from Theorem 3.11 in [69], and Theorem 1 in [56] proving the final CDAGs can only contain symbols of non-local variables. □

The set V of variables checked for equivalence may contain concrete values for some (e.g., `N=32`) and CDAGs for others (e.g., a reduction over a 32×32 input matrix). If all variables in V are concrete, such as in a test case, the proof holds only for these concrete values. However, non-local variables in V that are read before being written (e.g., function arguments, global variables), are automatically treated as symbolic [56]. *The equivalence result holds for any possible value symbolic variables may concretely take.*

3.6 Coverage and Limits

While we ensure tractability with α-deterministic programs as defined above, we do not capture the set of all valid deterministic programs. Typical unsupported constructs are atomic updates using e.g. an associative/commutative operation, as well as idempotent concurrent tasks that may execute at the same time. For this type of concurrency, tasks may execute in any order and yet produce the same result regardless of the order. But in our α-deterministic model, if there exist two possible schedules respecting all synchronizations that lead to a different ordering of read/write to a memory location, we fail, given we explore only one schedule to maintain scalability. For example, a `release` of 2 with two tasks performing an `acquire` of 1 at the same time will not be α-deterministic, irrespective of the operations being synchronized by these `acquires`. Efficiently supporting reorderable tasks is the subject of future work. This limitation does not prevent analyzing complex concurrency patterns such as multi-level buffering via FIFOs over multiple PEs using acquire/release locks in MLIR-AIE programs, as shown in Sec. 5.

Our system also remains limited by scaling. Although it achieves strong progress in time-to-solution versus traditional formal verification approaches, its linear time and space complexity with respect to the number of operations *executed* still leads to very large memory representations. In our experiments, the final memory state M may be >10GB for 100 million operations interpreted, and grows linearly with operation count. In a nutshell, we can handle matrix-multiplication of 256x256 matrices using about 10GB at peak to represent M, in a few minutes. But for larger problem sizes (eg 512x512) it may exceed 100GB.

Solutions to increased scaling of PEQC-MLIR focus on reducing the memory footprint to sub-linear space complexity. Folding computation trees dynamically into loop-based code producing the same CDAG if interpreted is a direction we are actively pursuing, with promising preliminary results: footprint for matmul 512x512 is reduced to about 1GB [69].

4 MLIR Lowering and Translation

We use a two-step approach illustrated in Fig. 2, and developed in this section: (1) convert an MLIR program to an equivalent one in terms of execution semantics, but using only a subset of core dialect operations; (2) translate the resulting MLIR program into our intermediate program representation. PEQC-MLIR is equipped with a restricted parser for the C language, and exposes an API via specific function calls in the program which are intercepted by the interpreter to model task spawns and the various synchronizations we support. Equipped with such a translator, we deploy existing LLVM MLIR lowering passes for specific dialects (e.g., `affine`) and develop our own MLIR lowering passes for the MLIR-AIR and MLIR-AIE dialects, presented in Secs. 4.3 onward.

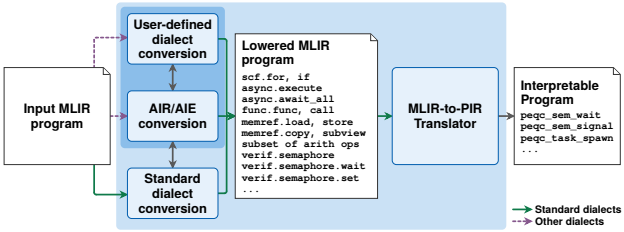


Figure 2: Conversion and translation to PIR

4.1 Translation to PEQC IR (PIR)

This stage takes as input an MLIR comprising only a subset of `func`, `arith`, `math`, `memref`, `scf` and `async`. It then translates it to a C-syntax AST, containing function calls to PEQC-MLIR's concurrent API when translating `async`. In addition, we expose the concepts of semaphores and their operations with a very small `verif` dialect, as shown in Sec. 4.3, to broaden the scope of MLIR programs that can be lowered to this common MLIR subset.

We note that the simple route of converting an MLIR program to C does not work in general: the `emitc` dialect and associated tools do not support the programs we target, especially when foreign dialects are used. Sec. 5.1 presents our verification of the AMD `AIR` dialect and its associated toolchain which is used to compile deep learning workloads to AMD AI Engines.

Translation of `arith` and `math` is trivial; we support 29 operations to date. `scf` and `func` are similarly trivial. In contrast, `memref` and `async` are more subtle. For `memref`, we emulate operations such as `copy`, `subview`, etc., with a pointwise loop iterating over all elements to be copied. If a `subview` cannot be eliminated with `-fold-memref-alias-op`, it is treated by creating a temporary variable, and inserting flushes to the original `memref` before the original is next accessed. For `async`, we translate tokens to `verif` semaphores and insert the corresponding `set`/`wait` as needed in operations that depend on the tokens, also producing a final semaphore when the `async` operation concludes for subsequent use. We encode `async.group` entries with distinct semaphores, one per member in the group. We do not implement a barrier construct,

but instead explicitly create as many point-to-point synchronizations as needed. While it may not be the most efficient, this approach is flexible and allows us to seamlessly model the asynchronous AIR and AIE dialects, OpenMP programs, etc.

4.2 Leveraging Existing MLIR Lowering Passes

To maximize the set of MLIR programs we can take as input, we leverage existing lowering passes provided with the `mlir-opt` standard LLVM toolchain. For example, `-convert-linalg-to-affine-loops` converts `linalg` operations (e.g., `matmul`) to an expanded MLIR `affine` program (e.g., the 3D loop nest for `matmul`). Then `-lower-affine` lowers the `affine` dialect to `scf` and `memref`, which we take as input.

To illustrate, our flow can take as input a `linalg` program, its tiled, optimized, and parallelized variant produced by the MLIR-AIR toolchain, and successfully lower them for eventual translation and correctness verification, as shown in Sec. 5.1. To achieve this, we designed our own conversion passes to standard dialects for two user-defined dialects: `AIR` and `AIE`.

4.3 Lowering of the MLIR-AIR Dialect

The MLIR-AIR dialect captures explicit asynchronous control, hierarchical parallelism, and dataflow between compute and memory, serving as a structured intermediate layer that bridges high-level abstractions and low-level accelerator code generation [73]. While some lowering passes are provided for standard MLIR dialects, user-defined dialects such as AMD's AIR and AIE must also be lowered to standard dialects. We developed these converters for a large subset of AIR and AIE constructs. Our open-source software [6] contains extensive documentation on operations we support, our approach to translation, and the complete C++ implementation of these converters (built using the LLVM MLIR infrastructure [5]). We illustrate below three constructs from the AIR dialect. Our implementation supports all AIR operations, albeit with some restrictions on arguments and options supported.

Conversion of `air.execute` The `async` dialect enables the description of sets of parallel tasks; tokens produced during task execution enable synchronizations between tasks spawned using `async.execute`. We convert `air.execute` to `async.execute`, a process that involves analyzing the `air.tokens` used for synchronization in the AIR program and converting them to groups of `async` tasks, and handling value results returned by parallel tasks. This is a form of lowering, as we encode the semantics of the source dialect operation via a new MLIR program fragment made only of standard operations that preserves the original execution semantics. In this case, AIR follows the semantics of an existing standard `async` operation, making conversion straightforward.

Conversion of `air.herd` Herds in the AIR dialect represent "a one or two dimensional array of adjacent AIE cores executing the same code region" [4]. In other words, one can create a parallel task for each AIE core modeled, carefully preserving the arguments to the herd (e.g., the L1 buffers manipulated by the herd) and synchronization tokens, and execute the same code block on each of them. This is equivalently represented using a carefully crafted `scf.parallel` construct.

Conversion of `air.channel` AIR Channels are used to communicate data and synchronize between concurrent entities. In contrast to the above examples, channels do not follow existing constructs in standard dialects. Channels can have blocking read/write operations (e.g., behaving like a blocking FIFO), can broadcast to multiple destinations, have special semantics on concurrent puts/gets on the same channel but to different buffers (implicit serialization in this case only), and can optionally perform on-the-fly data layout transformations of the data pushed to or pulled from the channel.

```
// An array of 4 x 4 streaming DMA channels
air.channel @chan_0 [4, 4] {channel_type = "dma_stream"}
// A streaming DMA channel broadcasting to 4 destinations
air.channel @chan_1 [1, 1] {broadcast_shape = [1, 4],
channel_type="dma_stream"}
// Receive a 4x4 tile into %dst from channel @chan_0
air.channel.get @chan_0(%dst[%c0,%c0][%c4,%c4][%c1,%c1]):
(memref<16x16xf32>)
// Asynchronous get with dependency on %t1
%t2 = air.channel.get async [%t1] @chan_1(%dst[%c8,%c0] \
      [%c4,%c4][%c1,%c1]) : (memref<16x16xf32>)
```

To implement conversion of `air.channel` to standard dialects, we developed a conversion using explicit `memref` buffers to store the data (and implement required data layout transformations, by explicitly copying into the proper layout into a new buffer). Unfortunately, `async` tokens can only be written once; this prevents modeling semaphores. For instance, to model a binary semaphore, one would need to create one token per *access* to the channel, a quantity immensely difficult to compute statically. To address this issue, we have developed a minimal `verif` dialect, dedicated to handling semaphores (`wait`, `set`, `acquire`, and `release`) as well as the type `verif.semaphore`. The `verif` dialect has direct conversion to the PEQC-MLIR API. Consequently, `air.channels` are converted to blocks with `memrefs` to store data, loops to copy data into a different layout, one semaphore per channel, and `waits`/`sets` around read/write operations.

4.4 Lowering of the MLIR-AIE Dialect

While MLIR-AIR focuses on higher-level constructs useful for representing computations on spatial architectures generally, MLIR-AIE is focused primarily on programming constructs that map cleanly to Neural Processing Units [3]. We have implemented support for the following `aie` operations: `buffer`, `core`, `device`, `dma_bd`, `flow`, `lock`, `mem`, `memtile_dma`, `next_bd`, `tile`, `use_lock`, `runtime_sequence` and the `objectfifo` operations: `createObjectFifo`, `link`, the pair `acquire`, `release`, and finally the `subview.access`. This is not the full AIE dialect; specifically, operations related to low-level routing (e.g., `packet_flow`, `switchbox`) are not currently implemented.

Concurrent MLIR-AIE constructs such as `aie.core` and `aie.mem` are translated to parallel tasks in the `async` dialect. Lock operations, such as `lock` and `use_lock`, represent counting semaphores and are lowered into the `verif` dialect. More complex communication constructs, such as `aie.objectFifo`, are lowered into a combination of buffer allocations, data copy operations, and semaphores. The main complexity here is representing complex data layout transformations that can be implemented as part of `aie.objectFifo` accesses.

Most other constructs lower into standard MLIR dialects representing sequential behavior, although some concurrent subtleties exist as discussed later in Sec. 5.2.

4.5 Trusted Codebase

We have formally proved the correctness of the algorithms for equivalence checking in PEQC-MLIR [69]. Overall, the correctness result produced by PEQC-MLIR is valid iff the whole software stack involved is itself correctly implemented.

The PEQC-MLIR system This includes PEQC-MLIR's parser, concrete expression evaluator, the implementation of hybrid concrete / symbolic interpretation, and the implementation of rewrite rule normalization. While the correctness of the interpretation algorithm was proven for a representative sublanguage [69], the implementation itself is not formally verified and must be trusted. This approach is similar to unverified tools used for equivalence checking and translation validation, for example Alive2 [48], CIVL [63] or KLEE [2, 20] which trust their implementation and SMT encoding.

MLIR lowering passes and PIR translation We rely on the correctness of existing `mlir-opt` passes such as `-fold-memref-alias-ops`, `-lower-affine`, and `-convert-linalg-to-affine-loops`. Unlike the optimization passes in Sec. 5.4, these are always applicable. We also rely on correct implementation of the standard MLIR operations `func`, `arith`, `math`, `memref`, `scf` and `async` into PIR. Operations in these dialects have typically a one-to-one match with operations in PIR.

AIR and AIE conversion. Rather than checking against existing semantics as for the trusted code above, our conversions from MLIR-AIR and -AIE to standard MLIR dialects *define* the semantics of the operations, as is common for MLIR dialects for verification [30]. We worked with the AIR and AIE developers to develop converters that faithfully capture the semantics of their operations, but there could still be mis-specifications, so this codebase is trusted as well.

5 Experimental Results

We now present experimental results on correctness verification of a variety of MLIR-based optimization flows. We present detailed case studies for AIR in Sec. 5.1 and AIE in Sec. 5.2, demonstrating the ability of PEQC-MLIR to verify optimized, high-performance implementations for the AI Engine, and to uncover potential subtle concurrency bugs. We then present extensive evaluation of `mlir-opt` on the PolyBench/C benchmarking suite [7], demonstrating PEQC-MLIR's ability to quickly find numerous known limitations but also unknown bugs in the tested toolchain in Sec. 5.4.

We implemented all our tools as open-source, BSD-licensed software[1] [6, 11]. PEQC-MLIR checks equivalence between a pair of programs using a single CPU core; below, we use a Debian 12.13 Linux server with an Intel Xeon E3-1240v6 running at 3.70GHz, with 32GB of memory.

5.1 Verification of MLIR-AIR Designs

To demonstrate the ability of PEQC-MLIR to verify pairs of programs using different MLIR dialects, Table 1 displays statistics for the evaluated program variants. All variants are produced as intermediate steps in the MLIR-AIR compilation flow and are eventually lowered to MLIR-AIE (further steps are evaluated in Sec. 5.2). Fig. 3 displays the flow of this correctness verification approach.

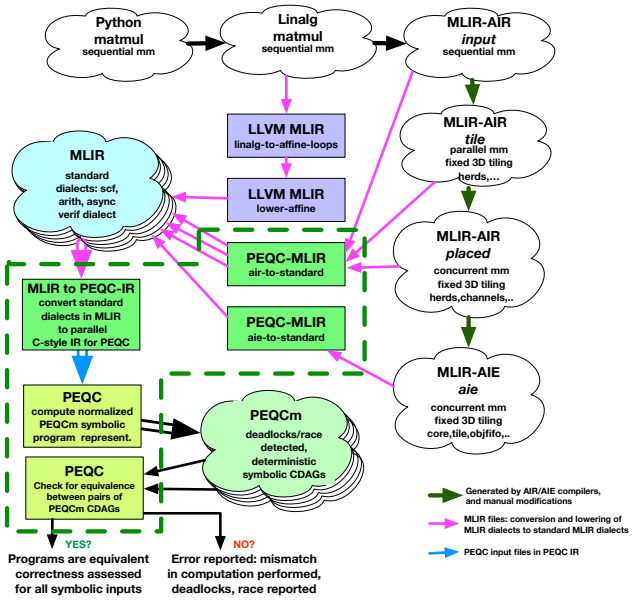


Figure 3: Flow of AIR/AIE Correctness Checking with PEQC-MLIR

The `linalg` program is built from a Python specification, and represents a sequential dense matrix-multiplication kernel, for a matrix (block) of size 32×32 to be run on the AI Engine. air.input, the reference input program to be compiled by the MLIR-AIR toolchain into a valid MLIR-AIE implementation on the AI Engine, was built by applying lowering of `linalg` to `affine` then to `scf`. air.tiled is a hierarchically tiled transformation of air.input, specified with a user-defined transformation script using the MLIR Transform dialect. It introduces AIR operations, such as `segments`, DMA copies, etc. air.placed is optimized and lowered from air.tiled, and introduces air.channels for communications.

These programs are verified pairwise as illustrated in Fig. 3. First, they are converted to equivalent standard MLIR dialects (using existing LLVM MLIR lowering passes, or PEQC-MLIR lowering passes we have developed), then translated to PEQC-MLIR's input Parallel Intermediate Representation (PIR) as per Sec. 4. These PIR files are then checked pairwise for equivalence, following Sec. 3.

Table 1: Statistics for AIR verification flow.

bench	#S	#L	nb-s	nb-c	nb-sy	hb-rat	t_{int}	t_{eq}
linalg	8	56	201870	0	0	0	1.61s	N/A
air.input	17	56	541420	0	0	0	3.39s	0.04s
air.tiled	102	160	743594	25	25	>0.00	5.56s	0.04s
air.placed	190	1030	1320799	440	77815	0.39	9.01s	0.04s

We report in Table. 1 metrics about the program source (#S: lines of code of original MLIR file; #I: for the final PIR file checked for correctness), and evaluation metrics (nb-s: total number of statements

[1] https://github.com/axolotls73/PEQC-MLIR

interpreted; nb-c: total number of tasks created; nb-sy: total number of synchronizations). hb-rat is the ratio between the number of nodes in our representation of the graph G divided by nb-s. Finally, t_{int} is the total time to interpret, that is to produce the final memory state for the program, and t_{eq} is the total time to check equivalence between two memory states, against that of the previous file. In this set of experiments, all files were proved equivalent, leading to the maximum equivalence time (the process ran to completion).

Interpretation time roughly follows the number of statements interpreted, which grows when lowering due to more fine-grain control-flow and additional data movements required between materialized hierarchical buffers. Equivalence time remains constant since the memory states for non-local variables are identical in these experiments (the programs are equivalent, for symbolic input matrices but a fixed problem size).

We have also used PEQC-MLIR to systematically explore tile size and herd size configurations with AIR. Verifying 720 variants for 64x64 problem sizes completes in about 20 minutes, running 16 verifications in parallel (one per core), for a peak memory usage of 10GB cumulatively. During development, we uncovered corner cases of missing synchronizations in the AIR program and improper local buffer manipulation when the air.launch size exceeds the herd size. These issues eluded testing: in practice, the herd size is used for on-machine testing, serializing parallel launches. All of these issues have been fixed.

5.2 Verification of MLIR-AIE Designs

PEQC-MLIR supports a rich subset of MLIR-AIE operations, as described in Sec. 4.4. We present a key case study, demonstrating the merits of PEQC-MLIR in catching subtle concurrency bugs, as well as the ability to verify equivalence between a low-level "hardware-friendly" optimized MLIR program and the original, sequential `linalg` program. In this flow, the final AIR program with channels and placement is lowered and optimized to an MLIR-AIE representation with `aie-opt`.

The final MLIR-AIE file is 753 lines of code, leading to about 2.2k LoCs for the final PIR file obtained after conversion and translation. In this case, PEQC-MLIR reported a possible race. Fig. 4 is an excerpt from the matrix-multiply MLIR-AIE program, focused on a chain of `acquire`/`release` operations for a task to execute on a PE (a memory tile). `aie.dma_start(S2MM, 0, ^bb1, ^bb5)` starts a DMA communication, in the S2MM direction, and both basic blocks `^bb1` and `^bb5` are started in parallel. In a zero-latency model (where only synchronizations can impose a relative ordering between instructions across different tasks running concurrently), this code leads to non-determinism: `bb4` may execute twice, or only once if `bb6` acquires `%lock_2_1_0` before `bb4` repeats. Note the `aie.dma_bd` operations in `bb4` and `bb6` do not implement the same movement due to different arguments. PEQC-MLIR reports this non-determinism, analyzing the file in less than 5s (it exits early on the first error, otherwise 11s for a corrected, equivalent version). This is a case where testing alone did not discover the concurrency error, which is unlikely to be triggered except in unusual but possible scenarios. The implemented fix [3] uses additional locks to ensure no two basic blocks can access the same buffer.

```
%memtile_dma_2_1 = aie.memtile_dma(%mem_tile_2_1) {
      %0 = aie.dma_start(S2MM, 0, ^bb1, ^bb5)
    ^bb1:  // 2 preds: ^bb0, ^bb1
      aie.use_lock(%lock_2_1, AcquireGreaterEqual, 2)
      aie.dma_bd(%buf20 : memref<64x16xi32, 1 : i32>, 0, 1024)
      {task_id = 0 : i32}
      aie.use_lock(%lock_2_1_0, Release, 2)
      aie.next_bd ^bb1
    ^bb2:  // pred: ^bb3
      aie.end
    ^bb3:  // pred: ^bb5
      %1 = aie.dma_start(MM2S, 0, ^bb4, ^bb2)
    ^bb4:  // 2 preds: ^bb3, ^bb4
      aie.use_lock(%lock_2_1_0, AcquireGreaterEqual, 1)
      aie.dma_bd(%buf20 : memref<64x16xi32, 1 : i32>, 0, 512,
      [<size = 64, stride = 16>, <size = 8, stride = 1>])
      {task_id = 0 : i32}
      aie.use_lock(%lock_2_1, Release, 1)
      aie.next_bd ^bb4
    ^bb5:  // pred: ^bb0
      %2 = aie.dma_start(MM2S, 1, ^bb6, ^bb3)
    ^bb6:  // 2 preds: ^bb5, ^bb6
      aie.use_lock(%lock_2_1_0, AcquireGreaterEqual, 1)
      aie.dma_bd(%buf20 : memref<64x16xi32, 1 : i32>, 8, 512,
      [<size = 64, stride = 16>, <size = 8, stride = 1>])
      {task_id = 0 : i32}
      aie.use_lock(%lock_2_1, Release, 1)
      aie.next_bd ^bb6  }
```

Figure 4: Excerpt of `acquire`/`release` non-determinism.

5.3 PolyBenchEq/MLIR

PolyBench is a benchmark suite of 30 numerical computations with static control flow, extracted from operations in various application domains (linear algebra computations, image processing, physics simulation, dynamic programming, statistics, etc.).

In this work, we establish a new version of PolyBench: PolyBenchEq/MLIR. This suite comprises 30 kernels, dropping ADI and durbin, and adding two convolution kernels extracted from Linalg to the original 28 PolyBenchs. For every benchmark, it includes the following variants. (a) A MLIR Affine version, extracted from the C version with PolyGeist, and proved equivalent with PEQC-MLIR (raising is validated correct). (b) A lowered scf version, using our MLIR lowering approach, also validated for correctness with PEQC-MLIR. (c) A collection of curated MLIR variants, generated by `mlir-opt` and others, as described in Sec. 5.4, classified in correct (validated equivalent by PEQC-MLIR) and incorrect (mismatch reported by PEQC-MLIR). In total, there are 30x2 + 300 MLIR files, enabling users to test PolyBench MLIR kernels for any purpose with *certainty* that the Affine MLIR code is correct with respect to the original C code. But it also enables testing and evaluation of correctness checking tools, by providing numerous non-equivalent versions of benchmarks, to enable detailed comparisons and evaluation of formal verification techniques on this problem space.

Table 2 presents the 30 kernels in PolyBenchEQ/MLIR and their descriptions, and P/L the number of parallel loops *as-is* in the kernel vs. the total number of loops, O is Y if the outer loop is parallel, P/P is the number of consecutive parallel loops as-is in the largest perfectly nested loop nest in the benchmark. Many linear algebra kernels are parallel/tilable as-is, and do not sufficiently exercise the capability of optimizers to detect legality of transformations. PolyBench uses scalars in many such kernels, requiring privatization for correct parallelization.

Table 2: 30 kernels in PolyBenchEq/MLIR

Benchmark	P/L	O	P/P	Description
2mm	2/6	Y	1/2	2 Matrix Mult. ($\alpha ABC + \beta D$)
3mm	3/9	Y	1/2	3 Matrix Mult. ($(AB)(CD)$)
atax	2/4	Y	1/1	Matrix transpose and vector mult.
bicg	1/3	Y	1/1	BiCG sub-kernel of BiCGStab solver
cholesky	0/4	N	0/1	Cholesky decomposition
conv2d	2/4	Y	2/4	2D convolution
conv3d	3/6	Y	3/6	3D convolution
correlation	5/9	Y	2/2	Correlation computation
covariance	4/7	Y	2/2	Covariance computation
deriche	1/12	N	1/2	Edge detection filter
doitgen	2/5	N	0/2	Multi-res. analysis kernel (MADNESS)
fdtd-2d	7/8	N	2/2	2-D Finite Diff. Time Domain kernel
floyd-warshall	0/3	N	0/3	Graph shortest path length
gemm	3/4	Y	0/2	Matrix-multiply ($C = \alpha AB + \beta C$)
gemver	5/7	Y	2/2	Vector mult. and matrix add.
gesummv	1/2	Y	1/1	Scalar, vector and matrix mult.
gramschmidt	4/6	N	1/1	Gram-Schmidt decomposition
heat-3d	4/7	N	1/3	Heat equation over 3D data domain
jacobi-1D	2/3	N	0/1	1-D Jacobi stencil computation
jacobi-2D	4/5	N	2/2	2-D Jacobi stencil computation
lu	2/5	Y	1/2	LU decomposition
ludcmp	0/9	N	0/1	LU decomposition + Forward Subst.
mvt	2/4	Y	1/2	Matrix Vector product and Transpose
nussinov	0/3	N	0/2	Dyn. programming for seq. alignment
seidel	0/3	N	0/3	2-D seidel stencil computation
symm	0/3	N	0/2	Symmetric matrix-mult.
syr2k	3/4	Y	0/2	Symmetric rank-2k update
syrk	3/4	Y	0/2	Symmetric rank-k update
trisolv	0/2	N	0/1	Triangular solver
trmm	2/3	N	2/2	Triangular matrix-mult.

5.4 Correctness Checking for MLIR toolchains

We now report extensive evaluation over 3620 MLIR program variants generated by production and experimental MLIR toolchains.

Experimental setup We use `mlir-opt` from LLVM19, LLVM21 and LLVM23[2] to generate transformed variants of PolyBench MLIR programs. The original PolyBench MLIR is built using Cgeist [8] from the PolyBench/C 4.2.1 C kernel [7]. We evaluate raising from C to MLIR, lowering MLIR dialects, and transformations on them by checking their equivalence using PEQC-MLIR. Verification time is proportional to the problem size evaluated: the entire suite can be run in 12 minutes for 630 variants, and uncovers the same bugs with, e.g., the mini and medium dataset sizes. Per-benchmark verification time, throughput of PEQC-MLIR over MLIR program variants, and scaling to large problem sizes is further discussed in [69].

Correctness of raising C to Affine MLIR Cgeist is the de facto standard for raising an affine program region in a C program to an `affine` MLIR program. We validated correctness of the raising by checking equivalence between the input C program and its raised MLIR version, and validated Cgeist for all *but* one benchmark: ADI (computation trees mismatch with the C version). We also evaluated raising programs with parametric loops instead of fixed loop bounds, and checked correctness for several dataset sizes, obtaining identical results.

Using `mlir-opt` *for affine loop transformations* `mlir-opt` offers numerous passes to optimize programs, in particular for the `affine` dialect. We evaluated `-affine-X` passes: `normalize`, `licm` and `unroll` are always-legal loop transformations, and should be applicable without error to any `affine` program. We also evaluated `loop-fusion`, `loop-tile`, `loop-unroll-and-jam`, each with and without parameters (e.g., tile size).

For all always-legal transformations, the result is always a correct program that respects the original semantics. On the other hand, our systematic testing over all PolyBenches uncovered a variety of incorrect codes generated, in minutes. Indeed, a misconception stems from typically testing with programs on which these transformations are always *valid* to apply, like on matmult: the loop nest is fully permutable, fully tilable, and has outer parallel loops, leading to correct code generated by these transformations. On the other hand, testing fusion on `2mm` generates wildly incorrect code, appending the second 3D loop nest as innermost in the first 3D loop nest. `floyd-warshall` is 3D-tiled, which is strictly incorrect, exposing the lack of legality check for these transformations. PolyBench exposes outer-parallel loops for most benchmarks, `mlir-opt` parallelization was correctly deployed (or not applied) in our tests.

To compound the issue, *we observe different results for different versions of MLIR*: progress has been made in LLVM21 vs. LLVM19. Typically, this is because the optimization application is restricted further in LLVM 21. There are also numerous cases where only 1 dimension is "tiled" (strip-mined) which is an always-legal transformation. Table 3 displays aggregated statistics for each LLVM version, showing the number of benchmarks where the pass did modify the code, how many are correct vs. incorrect and fixes/regressions from LLVM 19 to 21.

Table 3: Verification outcomes for `mlir-opt` passes on 30 PolyBench benchmarks for LLVM 19 and 21. LLVM23 results are identical to LLVM21 ones.

	Modified		Fail		Correct		Fix	Reg
	19	21	19	21	19	21		
(none)	0	0	0	0	30	30	0	0
`normalize`	12	12	0	0	30	30	0	0
`licm`	10	10	0	0	30	30	0	0
`unroll`	30	30	0	0	30	30	0	0
`fusion`	9	6	**4**	**2**	26	28	**3**	**1**
`unroll-jam`	4	4	**1**	**1**	29	29	0	0
`tile`	27	27	2	2	28	28	0	0
`parallelize`	21	21	0	0	30	30	0	0

We also tested pipelines with a combination of passes, where correctness narrows down to correctness on individual passes as displayed above. Overall, PEQC-MLIR enables a systematic evaluation *without the flaw of testing for specific input data only*, enabling users to quickly assess misuse of a toolchain, for example.

Correctness of Linalg lowering and optimizations We evaluated specifically the `linalg` dialect, which embeds computation primitives (e.g., `matmul`, `conv2d`, etc.) and transformation on them. Lowering from `linalg` to `affine` and eventually `scf` was evaluated. PEQC-MLIR validated the `to-loops` and `to-parallel-loops` variations of `-convert-linalg` against our baseline `to-affine-loops`, for several `matmul` and `conv2d` sizes. All cases were verified correct in our experiments.

[2]Commits `d022f6b`, `5e430af`, and `278dba3`, respectively.

Correctness of research tools Finally, we evaluate `polymer-opt`, a tool to automatically and correctly apply loop transformations using the polyhedral model [52], which performs all necessary dependence analysis of the input program and computes a legal transformation of it using the Pluto algorithm [18]. PEQC-MLIR exposed for example incorrect code generated for several stencil computations (jacobi-2d, fdtd-2d), with off-by-1 array accesses in the transformed program [69]. These were typically caught as computation tree mismatches by PEQC-MLIR. Other benchmarks were verified correct [69].

5.4.1 Outcome and Guarantees with PEQC-MLIR These experiments demonstrate the merits of fast and systematic formal correctness verification using PEQC-MLIR: it typically takes a couple minutes in our setup to evaluate all 30 PolyBench kernels with one optimization pipeline. PEQC-MLIR enables users to assess correctness of optimization pipelines by proving their equivalence, avoiding false positives that may arise from testing alone. Indeed, testing depends on using carefully built input data, to exercise the program fully. Emitting non-zero initializations is trivial and sufficient for some cases. But other affine benchmarks such as cholesky need the input to be a positive semi-definite matrix, otherwise the result is polluted with NaN values, as originally in PolyBench/C 3.2. PEQC-MLIR alleviates this problem, by considering the fixed-size input matrices *symbolic*: it reasons independently of any numerical value they would produce and is only concerned with the operations computing the final outputs.

PEQC-MLIR can quickly expose computation mismatches and parallelism errors that can originate from misuse of a tool (e.g., not realizing it applies an optimization without checking its applicability first) as well as actual bugs in toolchains. Interestingly, it can also validate codes produced by unstable research/experimental tools, enabling their safe deployment in production without the need for the entire tool to be robust. Note that, as discussed in Sec. 4.5, the correctness result computed by PEQC-MLIR holds only if the software stack itself is correctly implemented, including the lowering passes we rely on.

5.5 Other Approaches

Several powerful tools have been developed to assess the correctness of MLIR programs and their transformations, including translation validation systems [14, 76] and equality saturation approaches such as HEC [1, 81]. These tools also provide evaluation of a subset of PolyBench benchmarks, under various transformations such as tiling and unrolling. However, none support concurrent programs (e.g., operations from the `async` dialect), which limits the applicability on modern AI accelerator workloads which are typically parallel programs. In addition, these tools focus on the `affine` dialect and its transformations, under severe restrictions. In contrast, PEQC-MLIR relies on existing lowering passes for the `affine` dialect to standard `scf`, `memref`, etc. dialects, and therefore also seamlessly supports programs that do not originate from `affine` conversion, broadening significantly the scope of programs handled.

Overall, none of these tools have sufficient coverage to be compared against PEQC-MLIR with the experiments presented above: validating Cgeist raising requires supporting both C and MLIR, our experiments on Linalg parallelization and `mlir-opt` are also typically unsupported. Note that MLIR-TV requires explicitly unrolling all loops, and fails to terminate on the mini dataset size for the transformations we tested. HEC requires encoding the effect of transformations on the program, requiring extensions for every new transformation to be supported. However, interestingly, its complexity is not dependent on the number of operations executed by the program (PEQC-MLIR is), but on the syntactic program size (unlike PEQC-MLIR, which is mostly agnostic to this).

Although the HEC tool [1] does not support any PolyBenchEq/MLIR files as-is due to missing support for MLIR operations and dialects we use, for fair comparison we extended its parser and rules to evaluate `mlir-opt` variants in Table 3, eventually supporting 27/30 PolyBench kernels for self-equivalence (the none case in Table 3). HEC completes equivalence in 0.7s using about 40MB per benchmark for these, while PEQC-MLIR typically completes in 0.1s for the mini dataset. However, for more complex transformations such as the tilings we evaluate, it only supports 20 out of the 27 tiled variants, and fails to saturate for several buggy fusion cases. For transformations that significantly increase program size, such as a 16×16 unrolling on syr2k, its execution time reaches 51s for 213MB of memory [81]. Overall, HEC's robustness to larger problem sizes under simpler transformations is a trade-off with PEQC-MLIR's generalized support for complex transformations while scaling linearly with the number of operations *executed* by the program.

6 Related Work

Testing and fuzzing Testing is the dominant approach for compiler correctness due to its low cost compared to equivalence checking, and fuzzing generates inputs in clever ways to try to find compiler bugs more effectively [21, 49]. There are several fuzzers that target MLIR specifically, including MLIRSmith [74], which generates constrained random programs, MLIRod [64], which uses program dependences to constrain generation further, and DESIL [65], which can find silent bugs. While these fuzzing approaches can find bugs, they cannot prove the absence of bugs, that is, the correctness of the program.

Symbolic execution Symbolic execution is a technique for exploring feasible paths of a program via an SMT encoding, which is useful for finding inputs that exercise certain branches [20, 60, 80]. KLEE is a symbolic execution engine [2, 20], which operates on LLVM IR. UC-KLEE [57, 58] extends KLEE to reason about equivalence between two programs, and KLEE-Float [47] adds support for floating-point symbolic values. These approaches can in theory support equivalence between programs with parametric loop bounds, however in practice scaling limitations due to path explosion quickly occur for the type of workloads we target, even using concrete values for the parameters. PEQC-MLIR develops a form of hybrid concrete-symbolic execution, but its specialization to a single control-flow path avoids search entirely and allows it to scale to larger programs.

CIVL [63, 79] is an intermediate verification language with a symbolic execution engine which supports concurrent programs, for example those using OpenMP. However, it has similar scaling limitations: in practice, strong limitations about the problem sizes

and the number of threads are required to prove equivalence in realistic time, such as limiting the number of threads to 4 [79]. In contrast, the search-free approach of PEQC-MLIR does not require such strong size and thread count limitations. For example, for a `scf.parallel` loop nest, in PEQC-MLIR every iteration is assigned a different thread to capture the maximal parallelism that may be exploited eventually at run-time, handling thousands of threads.

Translation validation Rather than verifying the compiler, translation validation (TV) [54, 55] only verifies the compiler's output by proving the target program refines the source program. PEQC-MLIR also takes the approach of being agnostic to the implementation of code transformations. Unlike PEQC-MLIR, however, proof obligations for translation validation are typically discharged to SMT solvers via SMT encodings of program semantics.

Alive2 [48] excels at catching a wide class of intricate instruction selection and peephole rewriting bugs in LLVM IR. TV is also used to verify rewrites for specialized languages like Halide [24], and for high-level synthesis [22]. TV tools that target MLIR either create verification dialects that standard operations are converted to [30] or translate operations directly to SMT [14, 76]. However, transformations that significantly restructure the program (e.g., using channels for complex data communication in `MLIR-AIR`), can make proving equivalence intractable for current SMT solvers.

TV tools typically perform bounded verification, where loops are effectively unrolled to a fixed size. Intricate loop bounds that result from complex transformations for tiling may challenge these approaches. In contrast, PEQC-MLIR relies on concrete interpretation of these expressions, seamlessly handling arbitrarily complex loop bound expressions.

Equality saturation Equality saturation [68] is an approach to rewriting that models rewrites non-destructively, compactly keeping track of equivalence classes of terms during rewriting in a data structure called an e-graph. Modern implementations such as egg [78] and egglog [83] have made e-graphs practical for program optimization and verification [31, 77]. HEC [81] uses e-graphs to verify optimizations for MLIR, and is discussed in detail in Sec 5.5.

Program alignment One strategy for proving program equivalence is to align common corresponding features (e.g., loops or basic blocks) between programs to decompose verification conditions to make them easier for solvers. Alignments can be found by analyzing program traces [23], using e-graphs [25], or identifying common patterns [37, 38]. Alignment techniques are useful when programs are similar, but struggle to handle extensive transformations that add asynchronous concurrency and hardware constructs. PEQC-MLIR does not need to align program constructs, since it interprets the program and only compares final memory states.

Polyhedral methods The correctness of schedule-based transformations for polyhedral programs can be reasoned about statically [27–29]. But techniques to prove equivalence between two polyhedral programs have also been developed [15, 16, 36, 61, 72]. ISA [71, 72] proves equivalence between two affine programs, using parametric loop bounds. However, its time complexity quickly becomes prohibitive for numerous practical complex transformations [15]. Instead, PolyCheck [15] targets equivalence of programs without parameters, deploying dynamic analysis to check that iteration-reordering transformations preserve the original program semantics. In contrast, PEQC-MLIR does not require either program to be affine, only that their control and data-flow can be concretely evaluated, and is not limited to iteration-reordering transformations.

Proof assistants To guarantee a compiler is correct, the compiler itself can also be verified. CompCert [46] is a formally verified C compiler, and is extended to support concurrency in CompCertTSO [62] and compilation to a hardware description language in Vericert [35]. The verification in these approaches is specific to the passes implemented in the compiler, while PEQC-MLIR can also reason about untrusted, even hand-coded, program transformations.

Program properties can also be interactively proven in a proof assistant: program logics like VST [10, 12] and the Iris framework [32, 41, 67], can be used to prove rich properties of concurrent programs in Rocq [9]. Lean-MLIR [17] provides an MLIR frontend for Lean [53] and a framework for proving the correctness of peephole rewrites. These approaches can prove expressive properties, but rely heavily on manual proofs, while PEQC-MLIR is fully automatic and does not require any expertise in formal verification.

Concurrency verification Formal verification for concurrent programs is often used for properties broader than program equivalence. Model checking approaches [33, 42, 50, 70] explore possible interleavings to find concurrency bugs, and static and dynamic race detection approaches [13, 34, 66, 82] often, like PEQC-MLIR, build a representation of the happens-before relation to detect possible races. PEQC-MLIR does not support atomic operations or reorderable tasks, but it only needs to explore one schedule instead of many, which enables scaling for larger programs.

Prior works have reasoned formally about determinism for parallel languages, such as for X10 [45], Concurrent Collections [19], and Habanero-C with promises [40]. Our formalism in Sec. 3 took initial inspiration from these works, but we profoundly broadened the scope of concurrency and synchronization styles (e.g., binary and counting semaphores) that are proved to be deterministic, and also enable reasoning on symbolic variables.

7 Conclusion

Our proposed approach to correctness verification for a class of MLIR programs provides formal guarantees of correctness for a highly useful class of programs, in time linear with the number of operations executed by the program. The correctness of our approach is formally proven [69], and we demonstrated its potential for fast deployment and systematic compiler testing. Future work includes the automatic generation of MLIR converters and their automatic formal proof of correctness, and the development of symbolic equivalence techniques with sub-linear complexity.

Acknowledgements

This work has been supported in part by U.S. National Science Foundation award #2009020, and by an unrestricted gift from AMD. We dedicate this paper to the memory of Paul Feautrier.

References

[1] 2026. HEC: Equivalence Verification Checking for Code Transformation via Equality Saturation. https://github.com/Yu-Maryland/HEC

[2] 2026. The KLEE Symbolic Execution Engine. https://klee.github.io

[3] 2026. MLIR-AIE. https://github.com/Xilinx/mlir-aie

[4] 2026. MLIR-AIR. https://github.com/Xilinx/mlir-air

[5] 2026. MLIR: Multi-Level IR in LLVM. https://mlir.llvm.org

[6] 2026. PEQC-MLIR: Equivalence checking for MLIR programs. https://github.com/axolotls73/PEQC-MLIR

[7] 2026. PolyBench/C: the polyhedral benchmarking suite. hhttps://polybench.sf.net

[8] 2026. PolyGeist: C/C++ frontend for MLIR. https://github.com/llvm/Polygeist

[9] 2026. The Rocq Prover. https://rocq-prover.org

[10] 2026. Verified software toolchain. https://vst.cs.princeton.edu/.

[11] 2026. Zenodo artifact for PEQC-MLIR and associated experiments. doi:10.5281/zenodo.21878951 https://zenodo.org/records/21878952.

[12] Andrew W Appel. 2011. Verified Software Toolchain: (Invited Talk). In *European Symposium on Programming (ESOP)*. 1–17.

[13] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 53–62.

[14] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. 2022. Smt-based translation validation for machine learning compiler. In *International Conference on Computer Aided Verification (CAV)*. 386–407.

[15] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. PolyCheck: Dynamic Verification of Iteration Space Transformations on Affine Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[16] Denis Barthou, Paul Feautrier, and Xavier Redon. 2002. On the equivalence of two systems of affine recurrence equations. In *European Conference on Parallel Processing*. Springer, 309–313.

[17] Siddharth Bhat, Alex Keizer, Chris Hughes, Andrés Goens, and Tobias Grosser. 2024. Verifying Peephole Rewriting in SSA Compiler IRs. *LIPIcs, Volume 309, ITP 2024* 309 (2024), 9:1–9:20.

[18] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[19] Zoran Budimlic, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, and Frank Schlimbach. 2010. Concurrent Collections. *Scientific Programming* 18, 3-4 (2010), 203–217.

[20] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, Vol. 8. 209–224.

[21] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 197–208.

[22] Ramanuj Chouksey and Chandan Karfa. 2020. Verification of Scheduling of Conditional Behaviors in High-Level Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2020).

[23] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1027–1040.

[24] Basile Clément and Albert Cohen. 2022. End-to-end translation validation for the halide language. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–30.

[25] Robert Dickerson, Prasita Mukherjee, and Benjamin Delaware. 2025. KestRel: Relational Verification Using E-Graphs for Program Alignment. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 118:1073–118:1100.

[26] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2015. On characterizing the data access complexity of programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[27] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53.

[28] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem - Part I: One-dimensional time. *IJPP* (1992).

[29] P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming* 21, 6 (1992), 389–420.

[30] Mathieu Fehr, Yuyou Fan, Hugo Pompougnac, John Regehr, and Tobias Grosser. 2025. First-Class Verification Dialects for MLIR. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1466–1490.

[31] Oliver Flatt and Yihong Zhang. 2023. egglog In Practice: Automatically Improving Floating-point Error. In *E-Graph Research, Applications, Practices, and Human-factors Symposium (EGRAPHS)*.

[32] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31.

[33] Pavel Golovin, Michalis Kokologiannakis, and Viktor Vafeiadis. 2025. RELINCHE: Automatically Checking Linearizability under Relaxed Memory Consistency. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 2090–2117.

[34] Yizi Gu and John Mellor-Crummey. 2018. Dynamic Data Race Detection for OpenMP Programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, Dallas, TX, USA, 767–778.

[35] Yann Herklotz, James D Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal verification of high-level synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.

[36] Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. 2014. On program equivalence with reductions. In *International Static Analysis Symposium*. Springer, 168–183.

[37] Marie-Christine Jakobs. 2021. PatEC: Pattern-Based Equivalence Checking. In *International Symposium on Model Checking Software*. 120–139.

[38] Marie-Christine Jakobs. 2021. PEQCHECK: Localized and Context-aware Checking of Functional Equivalence. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 130–140.

[39] Hong Jia-Wei and Hsiang-Tsung Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. 326–333.

[40] Feiyang Jin, Lechen Yu, Tiago Cogumbreiro, Jun Shirako, and Vivek Sarkar. 2023. Dynamic Determinacy Race Detection for Task-Parallel Programs with Promises. In *37th European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 13–1.

[41] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of functional programming* 28 (2018), e20.

[42] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 96–110.

[43] Bernhard Kragl, Constantin Enea, Thomas A Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive sequentialization of asynchronous programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 227–242.

[44] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14.

[45] Jonathan K. Lee and Jens Palsberg. 2010. Featherweight X10: A Core Calculus for Async-Finish Parallelism. *SIGPLAN Not.* 45, 5 (Jan. 2010), 25–36.

[46] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *Embedded Real Time Software and Systems, 8th European Congress (ERTS)*.

[47] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F Donaldson, Rafael Zahl, and Klaus Wehrle. 2017. Floating-point symbolic execution: a case study in n-version programming. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 601–612.

[48] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 65–79.

[49] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[50] Eric G Mercer, Peter Anderson, Nick Vrvilo, and Vivek Sarkar. 2015. Model Checking Task Parallel Programs Using Gradual Permissions (N). In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 535–540.

[51] Jules Merckx, Alexandre Lopoukhine, Samuel Coward, Jianyi Cheng, Bjorn De Sutter, and Tobias Grosser. 2025. Eqsat: An Equality Saturation Dialect for Non-destructive Rewriting. *arXiv preprint arXiv:2505.09363* (2025). arXiv:2505.09363

[52] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to polyhedral MLIR. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 45–59.

[53] Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*. Springer, 625–635.

[54] George C Necula. 2000. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices* 35, 5 (2000), 83–94.

[55] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 151–166.

[56] Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, and Zhiru Zhang. 2024. Formal Verification of Source-to-Source Transformations for HLS. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. 97–107.
[57] David A Ramos and Dawson Engler. 2015. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security Symposium (USENIX Security)*.
[58] David A Ramos and Dawson R Engler. 2011. Practical, low-effort equivalence verification of real code. In *International Conference on Computer Aided Verification (CAV)*. 669–685.
[59] Grigore Roșu and Traian Florin Șerbănuță. 2010. An Overview of the K Semantic Framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.
[60] Daniel Schemmel, Julian Büning, César Rodríguez, David Laprell, and Klaus Wehrle. 2020. Symbolic partial-order execution for testing multi-threaded programs. In *International Conference on Computer Aided Verification*. Springer, 376–400.
[61] Markus Schordan, Pei-Hung Lin, Dan Quinlan, and Louis-Noël Pouchet. 2014. Verification of Polyhedral Optimizations with Constant Loop Bounds in Finite State Space Computations. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)*. Springer, 493–508.
[62] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)* 60, 3 (2013), 1–50.
[63] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.
[64] Chenyao Suo, Junjie Chen, Shuang Liu, Jiajun Jiang, Yingquan Zhao, and Jianrong Wang. 2024. Fuzzing MLIR compiler infrastructure via operation dependency analysis. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1287–1299.
[65] Chenyao Suo, Jianrong Wang, Yongjia Wang, Jiajun Jiang, Qingchao Shen, and Junjie Chen. 2025. DESIL: Detecting silent bugs in mlir compiler infrastructure. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 3093–3119.
[66] Bradley Swain, Yanze Li, Peiming Liu, Ignacio Laguna, Giorgis Georgakoudis, and Jeff Huang. 2020. OMPRacer: A Scalable and Precise Static Race Detector for OpenMP Programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–14.
[67] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A higher-order logic for concurrent termination-preserving refinement. In *European Symposium on Programming (ESOP)*. 909–936.
[68] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 264–276.
[69] Emily Tucker, Louis-Noël Pouchet, Erika Hunhoff, Stephen Neuendorffer, and Erwei Wang. 2026. Practical Formal Verification for MLIR Programs. arXiv:2605.01124 [cs.PL] https://arxiv.org/abs/2605.01124
[70] Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. 2010. Automatic verification of determinism for structured parallel programs. In *International Static Analysis Symposium (SAS)*. 455–471.
[71] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2009. Equivalence checking of static affine programs using widening to handle recurrences. In *International Conference on Computer Aided Verification (CAV)*. 599–613.
[72] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. on Programming Languages and Systems (TOPLAS)* 34, 3 (2012), 11.
[73] Erwei Wang, Samuel Bayliss, Andra Bisca, Zachary Blair, Sangeeta Chowdhary, Kristof Denolf, Jeff Fifield, Brandon Freiberger, Erika Hunhoff, Phil James-Roxby, Jack Lo, Joseph Melber, Stephen Neuendorffer, Eddie Richter, Andre Rosti, Javier Setoain, Gagandeep Singh, Endri Taka, Pranathi Vasireddy, Zhewen Yu, Niansong Zhang, and Jinming Zhuang. 2026. From Loop Nests to Silicon: Mapping AI Workloads onto AMD NPUs with MLIR-AIR. *ACM Trans. Reconfigurable Technol. Syst.* 19, 2, Article 28 (2026), 36 pages.
[74] Haoyu Wang, Junjie Chen, Chuyue Xie, Shuang Liu, Zan Wang, Qingchao Shen, and Yingquan Zhao. 2023. Mlirsmith: Random program generation for fuzzing mlir compiler infrastructure. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1555–1566.
[75] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.
[76] Yanzhao Wang, Fei Xie, Zhenkun Yang, Pasquale Cocchini, and Jin Yang. 2024. A Systematic Translation Validation Framework for MLIR-Based Compilers. *International Journal of Software Engineering and Knowledge Engineering* 34, 10 (2024), 1621–1640.
[77] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 12 (2020), 1919–1932.
[78] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* (2021).
[79] Wenhao Wu, Jan Hückelheim, Paul D Hovland, and Stephen F Siegel. 2022. Verifying Fortran Programs with CIVL. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 106–124.
[80] Fangke Ye, Jisheng Zhao, and Vivek Sarkar. 2018. Detecting MPI usage anomalies via partial program symbolic execution. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 794–806.
[81] Jiaqi Yin, Zhan Song, Nicolas Bohm Agostini, Antonino Tumeo, and Cunxi Yu. 2025. HEC: Equivalence Verification Checking for Code Transformation via Equality Saturation. In *USENIX Annual Technical Conference (USENIX ATC)*. 1181–1196.
[82] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. 2013. Array dataflow analysis for polyhedral X10 programs. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*. 23–34.
[83] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better together: Unifying datalog and equality saturation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 468–492.

A Artifact Appendix

A.1 Abstract

PEQC-MLIR is available at https://github.com/axolotls73/PEQC-MLIR. It is a tool to compute whether a pair of MLIR programs are semantically equivalent and free of parallelism bugs, reporting mismatches otherwise. It supports a variety of MLIR operations from the standard `scf`, `cf`, `math`, `memref`, `func`, `arith` and `async` dialects, as well as from the AMD `MLIR-AIR` and `MLIR-AIE` dialects. Other dialects may be conveniently lowered to this subset and processed by PEQC-MLIR, for example `affine` (via `-lower-affine`) and `Linalg` (via `-convert-linalg-to-loops`).

The artifact for this paper https://zenodo.org/records/21878952 is a snapshot of the PEQC-MLIR repository, that is all source code required to build PEQC-MLIR itself, and also includes self-contained docker installation scripts and all files required to reproduce experiments presented in this paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Program equivalence by hybrid concrete-symbolic interpretation.
- **Program:** PEQC-MLIR. and optionally LLVM21 with MLIR-AIR and MLIR-AIE support.
- **Compilation:** LLVM 21.
- **Run-time environment:** Docker on Linux for automated installation, compilation from source on a Linux x86_64 system also possible.
- **Hardware:** We recommend 50GB of disk space for the fully automated docker installation, and a x86 64-bit CPU with at least 4GB of RAM.
- **Output:** Statistics on the computation of symbolic memory state for both programs, whether they are semantically equivalent and free of parallelization errors, or why the interpreter aborted otherwise, including mismatches in the symbolic representation of the programs.
- **Experiments:** PolyBench/C raised with Cgeist, convolutions in Linalg, matrix-multiplication in MLIR-AIR and MLIR-AIE.
- **How much disk space required (approximately)?:** 50GB at most.
- **How much time is needed to prepare workflow (approximately)?:** 2h on an 8-core CPU.
- **How much time is needed to complete experiments (approximately)?:** 1h-2h on a single CPU core.
- **Publicly available?:** yes.

- **Code licenses (if publicly available)?:** PEQC-MLIR is an Open-Source BSD 3-clause software, the PAST interpretation engine is LGPL v3.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.21878952

A.3 Description

A.3.1 How to access This specific artifact is permanently available on https://zenodo.org/records/21878952, and also on GitHub in the `pact26-ae` branch: https://github.com/axolotls73/PEQC-MLIR/tree/pact26-ae. For the latest version of this software, including added coverage, bug fixes and performance improvements, we recommend directly using the main branch at https://github.com/axolotls73/PEQC-MLIR.

A.3.2 Hardware dependencies There are no strong hardware dependencies for this software, however we recommend a 64-bit x86 CPU, along with 50GB of disk space and 4GB of RAM. ARM processors may be used via Docker and software emulation of a x86 CPU.

A.3.3 Software dependencies Our self-contained Docker installation scripts assume a Linux host running a rootless installation of Docker; we validated the artifact on Docker 29.4.0 running on Debian 12.13. PEQC-MLIR can be compiled from source directly outside Docker, see https://github.com/axolotls73/PEQC-MLIR for installation steps. It assumes that LLVM21 (llvm21 branch) or LLVM23 (llvm23 branch) is installed with MLIR components. Optionally, for `MLIR-AIR` and `MLIR-AIE` support, the AMD AIR and AIE LLVM toolchain is needed (it is automatically fetched and built by the PEQC-MLIR installation scripts). Python 3.10 or later is needed.

A.4 Installation

Retrieve `peqc-mlir-docker-pact26-ae.tar.gz` from Zenodo at https://zenodo.org/records/21878952 and expand it, or navigate to https://github.com/axolotls73/PEQC-MLIR/tree/pact26-ae/experiments/pact26/docker, then build the Docker image:

```
$> ./image/build.sh
```

This script assumes a rootless Docker installation on 64-bit Linux. Alternatively, sources can be built natively on a Linux host following the installation procedure at https://github.com/axolotls73/PEQC-MLIR.

A.5 Experiment workflow

The Docker image `dockergen/peqc-mlir-2025-llvm21:latest` contains all software required (and more): a 2025 version of LLVM21 with `MLIR-AIR` and `MLIR-AIE` support, PEQC-MLIR, and Python packages. Note for updated versions of this toolchain, e.g. for the 2026 LLVM23 version of `MLIR-AIR` and `MLIR-AIE`, please see https://xilinx.github.io/mlir-air/dev/ and https://github.com/axolotls73/PEQC-MLIR. This artifact uses the LLVM21 2025 version.

We provide convenience scripts to manipulate this image. We recommend connecting an interactive container to this image with the provided scripts:

```
$> ./bin/dockergen-peqc-mlir-2025-llvm21-interactive
```

Note that exiting this container does not kill it: instead, rerunning this command reconnects to the existing container. It assumes a rootless docker install.

The first time a container is created, a few additional packages needed to run these experiments should be installed:

```
$> cd /data-host-share/PEQC-MLIR/experiments/pact26/
$> ./docker-first-setup.sh
```

The `/data-host-share` directory in the container is mounted from the host subdirectory `./data-host-share` and has read-write-execute permissions. This allows the user to exchange files between the container and the host seamlessly.

Every time an interactive shell is launched in this container (e.g., with `./bin/dockergen-peqc-mlir-2025-llvm21-interactive`), some environment variables should be set to run this artifact:

```
$> cd /data-host-share/PEQC-MLIR/experiments/pact26
$> source env-host.sh
```

A simple test can be run, to ensure everything works smoothly:

```
$> cd /data-host-share/PEQC-MLIR/experiments/pact26
$> ./run-example.sh
```

The expected output is:

```
=== [1/2] sequential matmul vs tiled matmul (expected: equivalent
    ↪ ) ===
YES, example/matmul-sequential.mlir and example/matmul-tile-and-
    ↪ parallelize.mlir are equivalent

=== [2/2] sequential matmul vs BUGGY tiled matmul (expected: NOT
    ↪ equivalent) ===
NO, example/matmul-sequential.mlir and example/matmul-tile-and-
    ↪ parallelize-buggy.mlir could not be proved equivalent

SMOKE TEST PASSED: equivalent pair verified, buggy pair rejected.
```

If this is not the output produced, there is a setup problem, and steps below are unlikely to work also. This script internally simply calls PEQC-MLIR on two pairs of programs, e.g.,

```
$> $PEQC_MLIR_ROOT/script/peqc-mlir.py --verbose --pastchecker-
    ↪ path=`command -v pastchecker` example/matmul-sequential.
    ↪ mlir example/matmul-tile-and-parallelize.mlir "A,B,C"
```

Where `PEQC_MLIR_ROOT` is the location of the root directory of the PEQC-MLIR sources, `pastchecker` is the PEQC tool itself, the two files to be checked are passed as argument, followed by a comma-separated list of variable names to be checked for equivalence (typically, the live-in/live-out variables).

A.6 Evaluation and expected results

Experiments presented in this paper can be reproduced via several push-button scripts. These scripts produce CSV files summarizing all results, and each individual test command line is displayed during script execution, for easy reproduction.

A.6.1 Correctness of Cgeist We evaluate Cgeist on PolyBench/C 4.2.1 from https://polybench.sf.net, testing fixed-size loop bounds and parametric loop bounds in the kernel, with values set in the test wrapper instead.

```
$> cd /data-host-share/PEQC-MLIR/experiments/pact26
$> ./run-all.sh cgeist
```

This will take a few minutes to complete. Expected failures: ADI and durbin, as reported in the paper. Expected outcome: `FAILED: 16`, `PASSED: 220`

A.6.2 Correctness of Linalg We evaluate several `mlir-opt` passes to optimize `Conv2D` and `matmult` expressed in Linalg, including loop parallelization flags, using `mlir-opt` from LLVM 19, 21 and 23.

```
$> cd /data-host-share/PEQC-MLIR/experiments/pact26
$> ./run-all.sh linalg
```

This will take about a minute to complete. Expected failures: none, as reported in the paper. Expected outcome: `FAILED: 0`, `PASSED: 18`

A.6.3 Correctness of `mlir-opt` *versions* We evaluate MLIR files produced by `mlir-opt` on PolyBench/C files correctly raised by Cgeist, with LLVM MLIR versions 19, 21 and 23. These generated files are stored in the artifact, we also provide scripts to regenerate them.

```
$> cd /data-host-share/PEQC-MLIR/experiments/pact26
$> ./run-all.sh  mini-initial-exp
```

There are 756 files per dataset size. Evaluating the mini dataset size takes about 1h, using a single CPU core. Note that a timeout is implemented in this script for each pair tested for equivalence. Expected failures for fusion, tiling and unroll-and-jam optimizations, as reported in the paper. Expected outcome: `FAILED: 38`, `PASSED: 718`.

A.6.4 `MLIR-AIR` *and* `MLIR-AIE` *experiments* We have provided a demo for `matmult`, from its `Linalg` sequential representation, conversion to various flavors of AIR and AIE, and their equivalence checking. This demo illustrates how the AIR/AIE files are lowered to standard MLIR dialects, then converted to PIR, and eventually checked by PEQC. *Please note the AMD MLIR-AIR and MLIR-AIE compilers are being actively developed and improved, this artifact uses a snapshot of this toolchain from 2025 and should not be used to produce AIR/AIE optimized files. Instead, please refer to the latest available version of this toolchain at https://xilinx.github.io/mlir-air.*

```
$> cd /data-host-share/PEQC-MLIR/experiments/pact26
$> ./run-airaie-demo.sh
```

This will take about a minute to complete. Press enter to move to the next step. Expected outcome: all tests pass except the AIE test, exposing non-determinism in a lock sequence, as illustrated in this paper.

A.7 Experiment customization

The above wrapper script `run-all.sh` simply invokes run.py with selected arguments such as the location of the destination CSV file, the type of files used as reference to check against, the timeout value, etc. Internally, it calls `peqc-mlir.py` as described above, to check a pair of files.

Refer to the extensive documentation in https://github.com/axolotls73/PEQC-MLIR for details on command line options, lowering passes and their coverage including detailed status for the `MLIR-AIR` and `MLIR-AIE` dialects, and convenience scripts available to run individual experiments.

A.8 Notes

This artifact covers experiments as presented in this paper published in October 2026, for perpetual availability and result reproduction. We however encourage the reader to use the latest available version of PEQC-MLIR on Github, as this software is under active development including feature extensions, bug fixes and performance improvements. PEQC-MLIR integrates installation procedures for the latest compatible version of the AMD MLIR AIR and AIE toolchain: https://github.com/axolotls73/PEQC-MLIR.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://pact2026.github.io/ae/